1069

Bell Laboratories Cover Sheet for Technical Memorandum

The information contained herein is for the use of employees of Bell Laboratories and is not for publication. (See GEI 13.9-3)

Title- **Introduction to Scheduling and Switching under UNIX**

Date- **October 20,1975**

TM- **75-8234-7**

Other Keywords- **UNIX**
Operating Systems
Timesharing Scheduler

Author	Location	Extension
T.M.Raleigh	**MH 2C-411**	**3390**

Charging Case- **49170-120**
Filing Case- **40952-001**

ABSTRACT

The UNIX timesharing system possesses a distributed supervisor. Selection of processes to use the CPU (process Switcher) and for memory use (the Scheduler) are separate functions within the operating system.

This memorandum discusses process Switching and Scheduling under UNIX using queuing models as an aid to visualization of the system's operation. The models are used strictly as an aid to understanding the system and no mathematical analysis is used.

The operating system uses a multiple queue feedback system for ordering processes for CPU usage. The concept of process priority as it relates to the queuing setup and the method by which the processor is relinquished is explained. Reentrancy within the operating system, together with its implications for interrupt processing and preemption are also discussed. Penalty schemes used for limiting the amount of time a process may use the CPU are discussed for both the current version of UNIX and the Research version of UNIX.

TM-75-8234-7

Pages Text 15	Other 18	Total 33	RIPEN,MFS IRMA B	CM
No. Figures 21	No. Tables 0	No. Refs. 23	MH2C548 SUBJECT MATCH	11/11/75 UNOS

E-1932-U (6-73) SEE REVERSE SIDE FOR DISTRIBUTION LIST

BELL TELEPHONE LABORATORIES, INC.

TM-75-8234-7

DISTRIBUTION
(REFER GEI 13.9-3)

COMPLETE MEMORANDUM TO

CORRESPONDENCE FILES

OFFICIAL FILE COPY
PLUS ONE COPY FOR
EACH ADDITIONAL FILING
CASE REFERENCED

DATE FILE COPY
(FORM E-1328)

10 REFERENCE COPIES

ALBERTS,SARBARA A
ARNOLD,S L
ARTHURS,EDWARD
BAYER,DOUGLAS L
BIRCHALL,R H
BIREN,MRS IRMA S
<BLINN,JAMES C
BLUM,MRS MARION
+BRANDT,RICHARD B
BURROWS,T A
CANADAY,HUDD H
CARDOZA,WAYNE M
CARRAN,J H
CARR,DAVID C
CHAFFEE,N F
CHODROW,MARK M
COOK,THOMAS J
COPP,DAVID H
COREY,D A
CRANE,RODERICK P
<CRUM,TED A
CUNNINGHAM,STEPHEN J
DE JAGER,D S
DOLGOTA,T A
DOWD,PATRICK G
DRUMMOND,R E
DWYER,T J
EDMUNDS,T W
ERRICHIELLO,PHILIP M
FACTOR,R M
FEDER,J
FIORE,MRS RHODA J
FLANDRENA,R J
FORTNEY,V J
FRANK,H G
<FREEMAN,K GLENN
GANNON,T F
GATES,G W
GLASSER,ALAN L
GRAVEMAN,R F
HAIGHT,R C
HALPIN,THOMAS
HANLITON,PATRICIA
<HANSEN,MRS G J
HUFKA,MRS FLORENCE
HYNAN,J
IVIE,EVAN L
JACKOWSKI,D J
JENSEN,PAUL D
JOHNSON,STEPHEN C

COMPLETE MEMORANDUM TO

JOHNSTON,WALTER E JR
KAPLAN,A E
KAUFELD,J C JR
KAUFMAN,LARRY S
KAYEL,R G
<KEESE,W M
<KELLY,L J
<KERNIGHAN,BRIAN W
KEVORKIAN,DOUGLAS E
KLEIN,MISS B L
LESSER,PETER V
LIND,R O
LORENC,ANTHONY
LUDERER,GOTTFRIED W R
LUDWIG,J J
LUKACS,M E
LYONS,T G
MACHOL,R E JR
MARANZANO,JOSEPH F
MASHEY,JOHN R
MORGAN,S P
NORTON,MRS SUZANNE W
O BRYAN,HENRY M JR
O CONNELL,T F
O NEILL,DENNIB M
OSBANNA,S F JR
PARA,P S
PATEL,C K N
PERDUE,R J
PEREZ,MRS CATHERINE D
PERNESKI,A J
PETERSON,RALPH W
PITTIT,MS GEORGETTE
PHILLIPS,S J
PILLA,MICHAEL A
RALEIGH,THOMAS M
RIDDLE,GUY G
<RITACCO,J E
ROBINSON,HERMAN E
ROCHKIND,M J
ROMITO,LETITIA J
ROSLER,LAWRENCE
SABSEVITZ,A I
SATZ,L R
SHRUM,EDGAR V
SIX,FREDERICK B
SMITH,D W
SOUTHERN,MS M J
STAMPFEL,JOHN P
STEVENSON,D E
STORM,A R JR
STURMAN,JOEL N
SWANSON,GEORGE K
SWIFT,R E
TAGUE,BARKLEY A
UNDERWOOD,R W
VASSALLC,J C
VIGGIANO,F A
VOGEL,D W JR
VOGEL,GERALD C
WANDZILAK,F D
WATKINS,G T

COMPLETE MEMORANDUM TO

WEBB,FRANCIS J
WEHR,L A
WHITE,RALPH C JR
WILSON,GEOFFREY A
WINHEIM,MISS IRENE A
WONSIEWICZ,B C
WOOD,J L
119 NAMEB

COVER SHEET ONLY TO

CORRESPONDENCE FILES

4 COPIES PLUS ONE
COPY FOR EACH FILING
CASE

ACKERMAN,A F
AHO,A V
AHRENS,RAINER B
ALCALAY,DAVID
ALLISON,CHARLES E
AMOSS,JOHN J
ANDERSON,MS K J
<ARCHER,RUSSELL E JR
ARNDT,DENNIS L
ARNOLD,GEORGE W
ARNOLD,THOMAS F
ARSENAULT,JAMES R
ASKEW,W H
BACCASH,MISS J M
BAKER,BRENDA S
BASELL,RICHARD J
BAUGH,C R
<BECKER,R A
BECKETT,J T
BELL,R E
BERNSTEIN,LAWRENCE
BERRYMAN,R D
BERRY,M J
BEYER,JEAN-DAVID
BICKFORD,N E
BILGOWS,RICHARD M
BLAZIER,S D
BLEICHER,EDWIN
BLY,JOSEPH A
BODEN,F J
BONANNI,LORENZO E
BORGWSKI,MRS JANE
BOTHUN,R H
BOURNE,STEPHEN R
BOWERS,J L
BOWLES,J B
BOYCE,W M
BROWN,COLIN W
BROWN,JAMES W
BROWN,W STANLEY
BROWN,WILLIAM R
BULFER,ANDREW F
BURNETTE,W A

COVER SHEET ONLY TO

BYRNE,EDWARD R
<CALESSO,GIULIO L
CAMPBELL,J H
CAMPBELL,STEPHEN T
CARROLL,J J
CASPERS,MRS BARBARA E
CASSETT,E W JR
CAVINESS,JOHN D
CHAMBERS,J M
CHAMBERS,MRS B C
CHANDRA,R
CHANG,S-J
CHEN,STEPHEN
CHERRY,MS L L
CHIANG,T C
CHIPPENDALE,R A
CHRIST,C W JR
CICON,J F
CIRILLO,CARL
CLAYTON,D F
CLOUTIER,J E
COBEN,ROBERT M
COCHRAN,MRS A
COHEN,HARVEY
COLE,LOUIS M
COLE,N O
CORMIER,ROGER J
CORNELL,B G
COSTON,WALTER P
COULTER,J REGINALD
CRAGUN,D W
CRUME,LARRY L
D ONOFRIO,MRS P L
D STEFAN,D J
DAVIS,D D
DAVIS,R D
DEUTSCH,DAVID N
DI PILLO,FRANK A
DICKMAN,B N
DIMMICK,JAMES O
DONOFRIO,L J
DOUGLASS,J J
DOWDEN,D C
DRAKE,MRS L
DUDLEY,MRS E H
DUFFY,FRANCIS P
EDELSON,D
EITELBACH,DAVID L
ELLIOTT,G L
ELY,T C
ESSERMAN,ALAN R
ESTOCK,R G
FABISCH,MICHAEL P
FAULKNER,R A
FELTON,W A
FETTE,CHARLES J
FISCHER,H B
FLEISCHER,HERBERT I
FOUGHT,B T
FOUNTOUKIDIS,A
FOX,PHYLLIS
FRANKS,RICHARD L

COVER SHEET ONLY TO

FRANK,MISS A J
FRANZ,ANN M
FRASER,A G
FREEDMAN,M I
FREEMAN,R DON
FREIDENREICH,MRS B
FULTON,ALAN W
GAREY,MICHAEL R
GARY,KAREN V
GAY,FRANCIS A
GEER,EUGENE W JR
GEPNER,JAMES R
GEYLING,F T
GIBB,KENNETH R
GIBSON,B T JR
GIMPEL,JAMES F
GITHENS,JOHN A
GLUCK,F
GOETZ,FRANK M
GOLASEK,MISS R
GOLDSMITH,L D
GOLDSTEIN,A JAY
<GORMAN,JAMES E
GRAHAM,R L
GREENBAUM,H J
GUERERA,DOMINIC J
HAFER,E H
HAGGERTY,J F
HAGGERTY,JOSEPH P
HALL,ANDREW D JR
HALL,JOE T
HALL,MILTON B JR
HARKNESS,C J
HARRISON,NEAL T
HARUTA,K
HAUSE,A D
HAWKINS,RICHARD S
HAYES,JERSMIAH F
HEATH,SIDNEY F III
HEDRICK,MISS ELLEN L
HELD,RICHARD W
HERGENHAN,C B
HEROLD,JOHN W
HESTER,S D
HOHIG,W L
HOOPER,E A
HOPPERT,D J
HOYT,WILLIAM F
HO,MS J
HUDSON,E T
<HUNCKE,D J
HUNNICUTT,CHARLES F
HYDE,J P
IERLEY,W H
IMAGNA,CLYDE P
IPPOLITI,O D
IRVINE,M M
JACKSON,JAMES H
JACOBS,H S
JESSOP,WARREN H
JOHNSON,DAVID S
JOHNSON,DAVID T

...
418 TOTAL

+ NAMED SY AUTHOR > CITED AS REFERENCE < REQUESTED BY READER (NAMES WITHOUT PREFIX WERE SELECTED USING THE AUTHOR'S SUBJECT OR ORGANIZATIONAL SPECIFICATION AS GIVEN BELOW)

MERCURY SPECIFICATION.....

COMPLETE MEMO TO:
8234

UNOS® = UNIX/OPERATING BYSTEM

COVER SSEET TO:
8231 8233

COTCOS = THEORY OF COMPUTER OPERATING SYSTEMS

HO CORRESPONDENCE FILES
HO 5C101

TM-75-8234-7
TOTAL PAGES 29

TO GET A COMPLETE COPY:

1. BE SURE YOUR CORRECT ADDRESS IS GIVEN ON THE OTHER SIDE.
2. FOLD THIS SHEET IN HALF WITH THIS SIDE OUT AND STAPLE.
3. CIRCLE THE ADDRESS AT RIGHT. USE NO ENVELOPE.

PLEASE SEND A COMPLETE COPY TO THE ADDRESS SHOWN ON THE OTHER SIDE.

Bell Laboratories

Subject: **Introduction to Scheduling and Switching under UNIX**
Case- 49170-120 -- File- 40952-001

date: **October 20,1975**

from: **T.M.Raleigh**

TM: **75-8234-7**

MEMORANDUM FOR FILE

This memorandum provides an overview of Scheduling and Switching under UNIX and discusses some of the basic timing within the operating system. A second more detailed memorandum[17] is forthcoming and will describe the algorithms used by the Scheduler and process Switcher and much of the details of their implementation.

1. Introduction

In order to provide a framework for discussion of the UNIX Scheduler and Switcher an overview of the operation of the system using queuing models will be used. No attempt will be made to arrive at any analytical results in terms of queuing theory as the mathematics are rather involved and not enough data is currently available about the distribution of the queues, distribution of arrivals in the queues, etc. for UNIX. (There are a number of sources in the literature [1,2,3,6,7,8,9,10,15,19,21] for those interested in queuing models.) The queuing model will simply be used as an aid to visualization of the overall operation of the system. Since the UNIX operating system possesses a distributed supervisor, it will also be a convenient method for illustrating how the supervisory functions of the system are spread throughout the system and how they work together.

While the mathematics of queuing theory will be studiously avoided in the following description, a number of topics such as the type and number of queues, service within queues, penalties and preemption must be fully discussed in order to completely construct a model. In order to do this, a very simple model will be elaborated and new aspects of the model will be introduced as needed.

It will be assumed that the reader is familiar in general with operating systems, with the basic functions of operating systems and with some of the basic terminology (processes, interrupts, traps, etc.). It is also not the purpose of this document to describe the operation of DEC hardware or peripherals[4]. There is ample literature available to provide a background for the reader[3,5,12,13,14,16,22,23].

A choice has been made in the method in which the operating system is viewed in this memorandum. The operating system could have been viewed as an entity which a user process communicates with when the process requires any services which it cannot provide for itself (I/O,etc.). The operating system performs the desired service and returns control to the user process. The second means of viewing an operating system is as a collection of shared code which implements services that a process may avail itself of. In this view, a request for service is viewed as a continuation of the execution of the process but in a different address space and with appropriate protection of other processes. The difference between the two is that in the first view the operating system is an active entity which satisfies a request, whereas

in the second view, it is a passive entity providing common code and protection for the continued execution of a user process. It is in this latter sense that the UNIX operating system will be discussed. This is appropriate as the UNIX operating system is a distributed supervisor; that is, the supervisory functions are not localized to one particular portion of the system or one particular process in the system. Rather, these functions are spread throughout the system. The chief advantage of this view is that it will allow us to speak about a "process relinquishing the CPU", making the distinction between roadblocking and preemption clearer.

2. Basic Concepts

2.1. Queuing Models

Figure 1 shows a simple queuing model with a *processor* and a series of *processes* queued up awaiting their turn to utilize the processor. These processes arrive in the system with some distribution and are selected to run on the CPU by some policy. This policy may be based on priority, execution time, bribing or any criteria that the system chooses and is usually referred to as the Scheduling algorithm.

Under the UNIX Operating System, the supervisory functions are distributed throughout the system and are not localized to one section of the system. The policy for use of the CPU is enforced by a function called the process *Switcher*. The *Scheduler* under UNIX refers to the part of the system that manages the selection of processes to be placed in or removed from memory. Although these are two separate operations, the fact that the Switcher can only select processes that are in memory for execution means that there is interaction between the two functions. Both of these functions must be understood in order to understand process execution behavior.

One of the earliest models studied in conjunction with timesharing systems is the Round Robin (RR) discipline (see Figure 2). Here each process is allocated a *quantum* of execution time and *fed back* to the end of the queue after it has executed for its quantum. Processes are selected on a first come first served basis and a process cycles through the queue using as many quantums as necessary to complete a job.

Numerous variations of this discipline have been implemented on past and present day systems. Figure 3 shows a queuing model with multiple feedback queues. Here we have a series of queues waiting for use of the CPU. Each queue could correspond to a grade of service or priority assigned to all processes in that queue. The higher the priority, the sooner the processes in the queue would receive service. A process could be assigned to a priority queue for the duration of time that it was in the system. The queue could instead be assigned based on the characteristics of the process. Each time a new process enters the system, it would be assigned to an appropriate queue based on its characteristics (JCL). Since the process characteristics relevant for queue assignment may not be known when the process enters the system or may change in time, a system could be *adaptive*. As processes execute in the system, more information is acquired so that the system can do a better job of queue assignment to achieve its overall service goals. We would therefore expect that a scheme whereby a process would move from queue to queue as knowledge about its characteristics is obtained or as its characteristics change is more efficient than one where a process is permanently assigned to a queue.

This adaptive approach is incorporated into the UNIX operating system. Basically, there are a series of queues representing fixed priority levels within the operating system. Processes within the system move from queue to queue depending on the type of activity they are performing.

There are a number of other features to the queuing model used by UNIX, however, to simplify matters we will assume for the moment that there are only a fixed number of queues at fixed priority levels.

The method for selecting a new process to execute on the CPU is a variation of the Round Robin method. A function within the operating system called the process *Switcher* continually selects processes from the queue with the highest priority that contains processes that can be run. Each process in a queue is marked to indicate whether it can be run or not. The Switcher may find several processes in a queue, but if none are runnable, the Switcher must look at a lower queue for a candidate. If no processes are fed back to higher queues the service within a queue will be Round Robin.

2.2. Priority Queues

UNIX uses both queues at fixed priority values and queues that are established at priority values as needed (floating queues). (These queues are discussed in detail in the succeeding memorandum[17].) The lowest fixed priority queue (PUSER - see Figure 3) contains most of the processes in the system. These are processes which have not been roadblocked and are awaiting their turn to use the CPU. The remainder of the fixed queues are for processes which have roadblocked or just become unblocked. These processes are placed in one of the higher queues so that the process Switcher will give them use of the CPU before the processes in the queue at PUSER. The numerical priority value assigned to these queues need not concern us at this time, however, the order that these priority values represent is a measure of how urgent the system deems that a particular activity should be serviced. The order of the most important queues in the system is,

1) PSWP - I/O to the swap device.
2) PBIO - I/O to a block device (disks, tapes, etc.).
3) PINOD - Access to an inode list.
4) TTOPRI - I/O to a character device (teletype, paper tape, etc.).
5) PUSER - A queue for processes that are executable in user mode.

Floating queues are established at fixed priority values in intermediate positions in the ordering scheme to solve problems of giving better service to a process and of penalizing processes that misbehave (see sections below).

2.3. Implicit and Explicit Queues

Without attempting to get into any of the details of implementation it should be mentioned that there is no elaborate system of linked lists within UNIX to explicitly implement the queueing structure illustrated in Figure 3. Instead, by the very fact that a process is assigned a priority, the queues implicitly exist. There is a tradeoff between the amount of linking and unlinking or the amount of searching that must be done in a system to implement either scheme and for the UNIX operating system the choice was made not to link together queues of processes at the same priority. This means that there is a large amount of searching required both when it is desired to choose another process to run and when a *wakeup* (see next section) occurs, however it does allow the establishment of an arbitrarily large number of queues with ease.

Now that the nature of the queues and the service discipline for selecting a process from a queue has been established, the determination of which queue a process moves to and when it moves to that queue must also be defined. The approach taken on UNIX is that when a process relinquishes the CPU (because it must wait for I/O completion, for access to a locked list,

etc.) it assigns itself a priority based on the reason that it blocked. This priority defines the queue that it will appear in when it is unblocked.

2.4. Semantics for Multiprocessing

If a process requests service from the system and the system cannot satisfy the request immediately, (I/O not complete, etc.) the process roadblocks itself. This is a signal for the process Switcher to run another process at least until resources are available for the first process. In order to implement this, the system must have both a means for indicating that the process is relinquishing the CPU and a means for indicating that the desired event has occurred. (It should be remembered that we are speaking in terms of a process continuing execution within the system when it requires service. While executing within the operating system, events such as I/O completion may be detected so that a mechanism for relinquishing the CPU can be implemented gracefully.)

Within the UNIX Operating System, a process relinquishes the CPU by executing the following function (within the operating system)

sleep(event,priority)

This marks the currently executing process as unable to execute until some *event* occurs. The quantity *priority* is the queue that the process will enter once it has been awakened. The *sleep* call results in the currently executing process giving up the processor and an attempt by the process Switcher to select and run another process. If none are available, the processor will idle until there are processes that can be run. When an event that unblocks a process occurs, a call of the following form,

wakeup(event)

awakens the process. The quantity *event* in both calls is a mutually agreed upon value (usually an address within the system) which is used to synchronize the blocking and unblocking of a process. This call makes *all* processes that were waiting for the occurrence of *event* runnable and places them on the queue that the sleep call specified. With this setup, several processes may block waiting for the occurrence of the same event but at different priorities. When the event occurs, all of the processes waiting on that *event* will be awakened (made runnable) but will enter different queues.

It should be mentioned that the UNIX operating system uses a procedure which is essentially the converse of the Multics Operating System convention[20]. Under Multics, a process relinquishes the CPU by a call of the form,

sleep(event)

and is awakened by

wakeup(event,priority)

Under this setup, the *awakening event* specifies the queue to which an unblocked process is to be placed. For UNIX, a process enters a queue at the specified priority *when it relinquishes the CPU* while for Multics a process would stay in its present queue until the wakeup occurs.

2.5. Feedback and Arrival

Now that the nature of the fixed queues has been established and a means for entering the queues has been given, a short example will be given to illustrate the transitions that a process might undergo.

So far, processes have only been discussed as existing quantities. Processes *arrive* in the system as a result of the FORK mechanism. This is done so that there is a parent-child relationship between processes in the system. This is the *only* means that a process may enter the system. (Another memorandum will discuss processes[18].)

A typical process would enter the system (via FORK) and be placed in the queue PUSER (see Figure 3) as a runnable process. At some time the process Switcher would select the process to run on the CPU. If during the execution of the process a system call is made and the process must relinquish the processor (e.g., to wait for I/O completion) the process relinquishes the CPU via the sleep call and enters one of the *higher* queues, say PBIO (for block device I/O). The process waits in the queue while the Switcher selects and runs other processes. The process cannot itself be chosen because it must wait for I/O to complete. When the I/O does complete, a *wakeup* will be sent to the process and it will appear as a *runnable* process on the queue PBIO. At some later time the process Switcher will again select the process to run (this will depend on how many processes are in higher queues) and the system call will be completed by the process. When the system call returns from the operating system to the user process, the process' priority will be changed to PUSER. This does not have much significance in this case because the process *has* the CPU and is not waiting in any queue to use it. (It will however be a factor when preemption is introduced.) Thus, the length of time that the process spends in the queue PBIO depends on how long it takes the event (e.g., I/O to complete) to occur. The time that the process spends as an active candidate in that queue depends on how many active candidates there are in higher queues and on how long these higher priority processes occupy the CPU's time. Figure 4 illustrates the example and shows the priority changes and the periods of time that the process is queued and running (The CPU time includes the time that the process spends executing within the operating system.)

3. Interrupt Processing

In order to continue with the elaboration of a model, a number of topics, such as the amount of time a process may use before being preempted, must be discussed. In order to illustrate exactly how the processor is relinquished and provide a basis for the specification of time quanta, some hardware issues must be developed. Preemption is discussed only after a few of these topics are explained.

The DEC PDP11 line of computers have a hardware Memory Management Unit which implements the process protection mechanism for UNIX. There are three modes that the processor may be in; User, Kernel and Supervisor (the last not available on 11/40's). For each of these modes there is a virtual address map in the Memory Management Unit which performs address relocation and contains access control permissions for blocks of mapped memory. Thus we can speak of the processor executing in User or Kernel mode (the Supervisory mode is currently unused by UNIX). Instructions exist for setting the address map and for fetching and storing across address spaces (MFPI - Move From Previous Instruction space, MTPI - Move To Previous Instruction space, etc.) however, the instructions for fetching and storing across address spaces are normally disallowed to user processes by the compilers and assemblers under UNIX so that the system is protected from user processes.

UNIX uses the Kernel Memory Management registers for mapping the virtual address space of the operating system and the User Memory Management registers for the address mapping of the executing user process.

The UNIX that resides in Kernel address space contains all of the code for satisfying the system calls as well as all of the code for the interrupt and trap handlers. When a user process makes a system call (via a trap) the processor will change states as indicated in Figure 5 from User to Kernel mode, perform the system call on behalf of the user and return control to the user process. All system services are performed while the processor is in Kernel mode.

3.1. Hardware Priorities

Interrupts are likewise handled in Kernel mode, however, as interrupts are asynchronous events, they may occur while the processor is in Kernel or User mode. Figure 6 shows a user process that is interrupted several times to process an interrupt. PDP11 computers have a hardware priority scheme which allows a limited amount of interrupt masking and stacking. There are seven different hardware priority levels that a device may interrupt the processor on. The processor also has a priority (in the processor status word) that may be set under software control to allow or disallow interrupts of lower hardware priority. Each device on the system is hardwired so that it interrupts the processor at *one* of the hardware priorities when it requires service. The interrupt handlers (software) in UNIX which reside in Kernel address space are not considered to be processes within the system as they are not scheduled by the software. They are essentially functions that are called as interrupts occur. Figure 6 illustrates a series of interrupts occurring while the processor is in User mode. The interrupts occur asynchronously, are handled in Kernel mode and may be stacked depending on the hardware priority of the interrupt. A case of interrupt stacking is also illustrated in Figure 6 where process A has been stopped to handle one interrupt when an interrupt of higher (hardware) priority occurs. The system handles the higher priority interrupt first, returns to handling the interrupt of lower priority, then returns execution to process A. Each time a process performs a system call or is interrupted, the system saves and restores the context of the interrupted process.

Under the UNIX Operating System the *processor's* priority is *always* set to *zero* when it is in User mode so that *any* interrupt occurring will be handled *immediately.*

3.2. Critical Regions and Mutual Exclusion

The case of interrupts occurring while the processor is in Kernel mode poses some serious problems for reentrancy to the operating system, so that there were a number of choices to be made in designing the system. Within the system there are a number of *critical regions* of code. During the execution of a critical region the system must prevent data that could be changed by interrupt handlers from being altered. These regions of code may also be critical regions because it is necessary to exclude other processes from accessing data or from executing the same code.

On some existing or proposed systems[5], this problem is solved through the use of special semantics for execution of these critical regions. The problem is solved by two separate methods under UNIX.

For the case where it is desired to prevent interrupt handlers from changing data within a critical region, the processor's priority is raised to a value that locks out interrupts from devices that could change the data. This is done by use of the *spl* (set priority level in the processor status word) instruction (simulated for 11/40's). A critical region of code within the operating system that required the lockout of devices producing interrupts at priority level 5 would be surrounded by the following two function calls.

```
spl5();
...
/*critical region software*/
...
spl0();
```

The first call raises the processors priority level to 5 (by executing the spl instruction) locking out all interrupts at that level and below for the duration of execution of the critical region. The processor's priority level is usually lowered to zero after this to allow any interrupts that have occurred in the interim to be processed. The spl0(), spl1(), spl4(), spl5(), spl6() and spl7() functions are available to set the *processor's* hardware priority to 0, 1, 4, 5, 6, and 7 respectively. Figure 7 shows an example of how the handling of an interrupt is delayed while the system is executing within one of these critical regions. The interrupt is delayed for the period shown and is processed when the processor's priority is lowered to a level *below* that of the pending interrupt.

Some regions may only be critical as far as certain interrupts are concerned. For example, executing a critical region in the character I/O region may require the lockout of interrupts from character devices but not from block oriented devices.

When several processes can access the same variables, a problem of *mutual exclusion* develops. This has implications for the design of an operating system, for if one process may be preempted in the midst of executing a system call any local or global variables accessed by that system call may be altered before the preempted process resumes. If it is assumed that the system clock or any interrupt handler can preempt the currently executing process, then when executing a critical region of this type it is necessary to prevent these interrupts from occurring. (This situation could be avoided if there were some special hardware instruction or programming semantics for this situation but as there are none available under UNIX, the best solution would seem to be to prevent the processor from handling interrupts during the execution of these regions.) This is, however, not desirable as the clock interrupt handler makes interrupt request at hardware priority 6 and locking out the system clock would also lock out interrupts from *all* other hardware devices. Critical regions would then have to be limited to a section of code that could only execute for a short period of time (otherwise timeouts on the UNIBUS would occur) and the total number of such critical regions would also have to be limited.

The solution chosen by the designers of UNIX was to allow preemption of the processor in Kernel mode only at the end of a system call (that is just before the system returns control to the user process) or at the end of processing an interrupt. This essentially eliminates the reentrancy requirement for the operating system.

For the interrupt handlers, the approach has also been taken that interrupts for a particular device are handled by the processor at the priority that they occur. That is, the processor's priority is set to the same level as that of the interrupt thus locking out all other interrupts of the *same level and lower.* This eliminates the reentrancy requirement for interrupt handlers. As a beneficial side effect, both of these design choices eliminate the need for doing additional context saving (Kernel mode registers would need to be saved in a reentrant system).

3.3. Preemption

Under the UNIX Operating System, a process is preempted by the system clock or as the result of some interrupt which wakes up a process. In the case of the system clock, the preemption is done to enforce a time limit on the execution of a process. For interrupts which wakeup another process, the preemption is done in an effort to give service to a process that has willingly relinquished the CPU as soon as possible. The speed at which it will receive service depends on how urgently (what priority queue) the process should be served and on how many processes there are in higher queues. Because the system is non reentrant for the reasons discussed above, preempting a process that was executing in User mode and one that was executing in Kernel mode is different and will be discussed separately. Preemption as the

result of an interrupt waking up a process will be distinguished from the system clock preempting a process even though preemption by the system clock is the result of an interrupt (the clock interrupt). The reason for this is that for ordinary interrupts, the preemption is done to give reasonable response to processses that willingly relinquish the CPU while preemption by the system clock is part of a penalty scheme to limit the use of the CPU.

We will first consider the case where the system clock or more specifically the clock interrupt handler preempts a process. Figure 8 shows a process executing in *user mode* when a clock interrupt occurs. The context of process A is saved when the clock interrupt occurs. The interrupt is processed and in the case shown, the clock interrupt handler has decided to preempt the process so that the Process Switcher runs and selects a new process. Since the clock interrupt handler generates an interrupt once every sixtieth of a second, it would be unwise to preempt a process every time a clock interrupt occurred when the processor was in user mode. With an average instruction execution time of approximately 2.5 usec. for 11/45 processors, a 16.6 msec. time slice between process switches would not allow much computing. (The question of time quanta is taken up in a later section however, the system clock will preempt a process only at the one second clock interrupt.) All processing of the interrupt occurs before the clock interrupt handler preempts the process so that there is no problem with reentrancy.

As discussed previously, when the processor is in User mode all interrupts are allowed. Figure 6 shows interrupts being processed and control returning to the user process. If an interrupt results in some other process being awakened, then rather than return control to the same process after the interrupt is handled the process Switcher is called. Figure 8 illustrates the sequence of events. Again, the determination of whether to run the process Switcher and preempt the previously executing process is made only after the interrupt is processed and control is about to be returned to the user process.

Figure 9 shows a process which has made a system call and is executing in Kernel mode when an interrupt occurs. If a process could be preempted in the middle of a system call, the problem of reentrancy discussed previously could produce problems of mutual exclusion if the system call was not coded as a reentrant function. By disallowing preemption during this period the reentrancy problem is solved. For this reason, the one second clock interrupt is not allowed to preempt a process if the clock interrupt occurred while the processor is in Kernel mode.

For interrupts (which wake up some process) occurring while the processor is in Kernel mode, it is desirable to be able to preempt the current process even though the interrupt occurred in Kernel mode. Preemption is still disallowed by reentrancy requirements, however, the system can remember that an interrupt occurred while the system call was in progress so that the process Switcher can be called when the system call is finished and the process preempted at that time. Figure 10 illustrates this case.

In summary, the types of preemption that may occur are:

1) As the result of the system clock preempting a user process when it is executing in User mode to enforce a time limit.

2) As the result of an interrupt which wakes up a process.

   a) If the interrupt occurs in User mode the interrupt is handled immediately and the preemption occurs after the interrupt is processed.

   b) If the interrupt occurs while the processor is in Kernel mode, the currently executing process is preempted at the *end* of the system call it was executing.

3.4. Traps

Traps out of user mode are handled in exactly the same manner as interrupts (system calls are made by trapping). Traps occuring while the system is executing in Kernel mode are not normally possible. If they occur, they are regarded as serious hardware errors and the system is brought down as gracefully as possible.

4. Timing and Penalties

4.1. Time Quanta

So far, the manner in which the processor is relinquished has been discussed from two standpoints. Either it was relinquished willingly as the results of some resource not being available or it was *preempted* by the system clock or as the result of an interrupt waking up another process. Preempting a process when a wakeup is issued by an interrupt handler is done so that when a process roadblocks at a high priority (disk I/O,file access,etc.) will receive control as soon as possible after the resource is available. There will however be processes in the system that are *compute bound* or *system bound* and will not willingly relinquish the CPU. *Compute bound* processes can be identified as those processes which spend most of their time in user mode, never making any system calls and can easily be preempted since no reentrancy requirements are violated. *System bound* processes are processes which spend most of their time making system calls *which do not compete for system resources* and hence never willingly relinquish the CPU even though they spend a conderable amount of time in the system. These processes are more difficult to isolate. An example of this would be a process in a tight endless loop requesting the time of day from the system. In order to even out the performance of the system so that each user will see approximately the same type of response, it is necessary to limit the amount of time (a quantum) that a process may use the processor at one time.

The present version of UNIX depends on an averaging type of effect to enforce a one second time quantum limit on processes. Once every second the clock interrupt handler determines whether the clock interrupt occurred while the processor was in User or Kernel mode. If the processor was in User mode then the currently running process is preempted after the fashion shown in Figure 8. The assumption is made that the process has run for a full one second quantum. This may not be the case but on the average in will not damage the response of the process. If however, the clock interrupt occurred out of Kernel mode, no preemption to enforce the time limit is made because of reentrancy requirements and because there is a finite probability that the process will roadblock anyway as the result of the system call. (This is not strictly true as the clock interrupt handler does behave as other interrupt handlers and may wake up other processes. In particular, the clock interrupt handler maintains an *event* called the *lightning bolt*. The clock interrupt handler issues a wakeup to all processes sleeping on the lightning bolt once every four seconds. In this case, the clock interrupt handler is not directly preempting a process, rather, the same mechanism as was employed for the interrupt handlers when they issued a wakeup to a process is employed.) Figure 11 illustrates the effect of enforcing the one second time limit (on the average). The behavior of the processes (number of system calls, time to process a system call , etc.) has been greatly exaggerated for clarity of the drawing. Note that if it is assumed that the system calls take only a short amount of time and that processes spend much of their execution time in user mode the one second time slice will be enforced. Also, for the case where the clock interrupt occurs out of Kernel mode, the system call that was being executed may result in the process blocking so that the likelihood of the one second time quanta maximum will be increased.

The time quanta criteria will only influence compute bound jobs which run on the average for their full quantum. This however, is not a sufficient penalty to apply because an I/O

bound process will relinquish the processor a large number of times, thus giving the compute bound processes a large number of full quanta to use. Some other mechanism is required to insure that compute bound processes do not hog the CPU. To provide this restriction the time quantum criteria is coupled to a penalty scheme.

4.2. System Call Limitation

The preemption applied at the end of the one second interval limits the amount of time that a compute bound process may occupy the system's time. The preemption is not applied if the CPU is in Kernel mode so that system bound processes would not be detected. (The process could have been preempted at the end of the system call as is done for interrupts awakening processes, however, this would be an additional penalty applied to normally interactive processes which relinquish the CPU when they make a system call. That is, a process might only have made one system call when it was preempted.) A count of the number of consecutive system calls between a process roadblocking itself is kept. If a process makes seventeen consecutive system calls without having to roadblock, the process is considered to be behaving as a system bound process and is preempted.

4.3. Process Penalties and General Amnesty

In order for the preemption schemes of the previous section to be effective, they must be coupled to a penalty scheme that takes into account the priority queues. The model presented so far possesses a number of fixed priority queues awaiting use of the processor. The queues at higher priorities than PUSER are for processes that have relinquished the CPU and are waiting to be awakened or serviced. The duration of time that a process spends in one of these higher queues is typically short. (Only a very interactive process would spend a good deal of time in the higher queues and most of that time the process would not be runnable.) The PUSER queue would then contain the remainder of the processes in the system. In particular, the compute bound and system bound processes would always appear in this queue because they do not relinquish the processor of their own volition. If we were to discount for a moment any processes in the other queues, then a compute bound process in the PUSER queue would receive extremely good service (because it would run for its full quantum) compared to processes which were less compute bound in that queue. This assumes that the service within a queue is Round Robin. Compute bound process would receive a large percentage of the CPU's time relative to interactive processes. The same would be true of system bound processes; even though they are preempted after a certain number of system calls, they would achieve a higher percentage of CPU time compared to processes that roadblocked. In order to build into the system a certain amount of adaptivity and smooth out the response per user, a scheme whereby processes receive a *priority* penalty for poor behavior *when they are preempted* is used. This in effect corresponds to the formation of new (lower priority) queues as needed to take care of these processes (see Figure 12). Thus if the process Switcher used the scheme where higher priority queues are served before those lower priority queues, we would have a Shortest Elapsed Time (SET) effect for a group of compute bound processes (see Figure 13). Once a process had used its quantum, it would be placed in a lower priority queue. Since selection of the next process to run is done on a highest priority queue basis, an execution profile of the system would show that those processes which have received few quantums would be favored over those which have had many quantums. For compute bound process, those with few quantums would quickly receive enough quantums to catch up to those which had received a higher number of quantums. Processes which were less compute bound would find themselves favored as they would continually appear in higher queues.

It should be reemphasized that throughout this discussion of penalty schemes we are referring to processes that *do not* behave as the normal timesharing job. Interactive processes should escape the effect of any penalty scheme in the long run. Those processes which are misbehaving will be found in the queue PUSER (or lower queues) never in any of the fixed priority queues. Queues above this contain processes which have recently relinquished the CPU and are waiting for resources in order to be served.

Coupled with the preemption scheme discussed in the previous sections, there are two different penalty criteria that are applied to prevent processes from obtaining more than their fair share of the processors time.

One penalty scheme is intended to penalize system bound processes. At the end of every seventeen system calls, a process's priority is lowered by one point and it is preempted if none of the seventeen system calls resulted in the process roadblocking. Applying the preemption after the seventeenth consecutive system call is the criteria used to isolate the system bound processes from the more interactive processes. When the (system bound) process is again selected to run, a *general amnesty* is declared for that process so that it will return to its former priority (PUSER). Figure 14 illustrates the priority variations of a process which continually spends its time in system space without ever having to block. Each of the vertical ticks on the normal user priority line (PUSER) represents both the one time imposition of the one point penalty and the preemption of the process.

A similar scheme is utilized for penalizing compute bound processes. For each time that a process is found executing in user mode when the *one second* clock interrupt clock interrupt occurs, a one point priority penalty is imposed on the process. As discussed previously the clock interrupt handler should on the average provide the one second time quantum slicing, so that this penalty corresponds roughly to a penalty for using the full one second quantum. Unlike the penalty for system bound processes, this penalty is cumulative so that a process will continue to be penalized if it continues to exhibit the same type of behavior. Currently, the lowest queue to which a process may sink is set at 105 (5 queues below PUSER - see Figure 15). The queue at 105 corresponds to a process which has received five quantums without requesting *any system service.* As with the previous scheme, a *general amnesty* is declared as soon as the process stops behaving as a compute bound job (i.e., when it makes its first system call).

In both of the cases discussed above, a *general amnesty* was declared as soon as a process began to exhibit normal (timesharing like) behavior and there was a limit on how undesirable a process may become in the eyes of the system. This is done so that no job reaches a state where it is so undesirable that it receives *no service* or excessively poor service (e.g., a third shift job in the batch world).

4.4. New Penalty Scheme

The penalty scheme described above is somewhat blunt in that at least for the compute bound case it relies on identifying the CPU hog based on the fact that on the "average" he will be found executing in user mode. This however takes time to determine and is something which is dependent on the number of other processes in the system, the type of activity, number of interrupts, etc. For processes that hit on the right combination of misbehavior, the limits and penalties will be escaped and the process will use a good portion of the CPU's time. In order to more accurately and quickly detect misbehaving processes, a scheme whereby the cumulative execution time of each process is kept has been adopted on the UNIX Research System. (The clock interrupt occurs once every sixtieth of a second so that if one assumes that all of the CPU time since the last clock interrupt was occupied by the process that was interrupted an accurate account of the CPU usage may be kept. It is impossible to discount the time that was used to handle the clock interrupt or for any interrupt that occurred between clock interupts.) Priority penalties for a process are applied based on the number of

cumulative CPU seconds the process has received without roadblocking. A one point priority penalty is applied for each of the first five consecutive CPU seconds and one point for each fifteen seconds thereafter. Figure 16 illustrates the priority penalty scheme. (A continuous function is shown for clarity however it should be remembered that the graph should actually be a step function as in Figure 15 .) There is no longer a distinction between the penalty for CPU bound and system bound processes. Note also that the penalty is rather severe at first (identification of misbehaving process) but softens (to prevent total discrimination) after the first five seconds. There is no immediate ground level reached either as with the previous scheme. The process will continue to be penalized until it reaches the floor value for priorities (priority 127 see Figure 16). As with the previous scheme, a general amnesty is applied as soon as the process begins behaving properly.

4.5. User Initiated Penalties

There is a system call (NICE) available which allows a user process to lower the base value (PUSER) of its priority and thereby voluntarily receive a lower grade of service (effectively run in the background). This essentially just shifts the base line at which the penalty scheme is applied to the process. It does *not* in any way affect the process's placement in any of the fixed queues in the system, (PINOD, PBIO, etc), so that the response to a request for a system resource is the same. The only difference is that because of the downshift of the base user priority the process will receive a smaller fraction of the CPU's time.

4.6. Achieving a Better Grade of Service

The same system call which allowed a user process to lower its base priority level, will allow a process with super user permissions to achieve a better grade of service. This is done in exactly the same manner as was used to lower a process's base priority. In this case, however the base value is raised. This is a crude tool for obtaining better service because as higher and higher values are chosen for a process's base priority, competition with the fixed queues in the system becomes a factor (see Figure 16) and the response to system calls by other processes is degraded. For a typical well behaving process, the effect of the nice system call is to establish a new queue above PUSER for the "special" user process. This "special" process will occupy the system's time and only during the periods that it is roadblocked will processes of lower priority be served. In the case of CPU bound or system bound processes, raising the base priority dramatically changes the system's performance since the CPU bound processes will use all of their time quantums and the penalty scheme will take some time before the process's priority is lowered to a point where other processes can compete for service. With a general amnesty declared, processes which are not completely compute bound have an even greater effect. For the CPU bound portion of their execution they will be prime candidates for the process Switcher and since they are partially interactive, they will roadblock occasionally restoring them to the high priority that the NICE system call gave them.

5. Major Divisions in the System

Now that the details of the model have been clarified, a more complex model can be described and the domain of the process Switcher and the Scheduler may be illustrated.

5.1. A Complex Model

In order to more fully describe the distribution of function within the operating system, the queuing model of Figure 3 will be expanded and replaced by that of Figure 18. Here, we see that several additional queues and feedback paths have been added.

Two new queues have been introduced. The first queue is labeled WAKEUP and consists of all those processes which are *in memory* but are *blocked* awaiting some resource. When they are awakened they are fed into the proper queue. In the previous model we assumed that the process Switcher knew which processes could be run and which could not. This is still the case, however, conceptually it is clearer to view these unrunnable processes as being in a queue of their own. The WAKEUP queue can be considered to be a kind of free running queue as processes make the transition to one of the priority queues at the time they are awakened (i.e., asynchronously). Note that processes which roadblock themselves (by calling sleep (event, priority)) thereby placing themselves on the WAKEUP queue are fed back only to the fixed system queues, never to a user queue. This is because, as discussed previously, a process goes to sleep at one of the fixed priority levels (i.e., user processes cannot call the sleep function directly) as the result of a system call. The second queue (CORE) consists of all of those processes on the swap area. These processes may be sleeping or executable, however, they cannot execute until they are brought into memory.

An extra path has been added to show the path taken by processes that are preempted. These processes are runnable when the CPU is taken from them so they are fed back directly to one of the queues. These processes are fed back only to the user queue (PUSER) or to one of the penalty queues.

5.2. The Process Switcher

The process Switcher is the function within the system which selects the next process to run from the queuing setup described previously. The processes present in these queues are only those processes which are *not* roadblocked and which are *in memory*. The Switcher knows *nothing* about the size or arrangement of memory. Its sole function is to find the runnable processes in the highest queue and to select one to use the CPU. The length of time that each process will run at one time will depend on the process' demands on the resources of the system and their availability and will be subject to the time quantum limits discussed earlier. If none of these processes are fed back to higher queues, the service within the queue will be Round Robin.

5.3. The Scheduler

The Scheduler is itself a process (albeit a process *within* the operating system that runs entirely and continually in Kernel mode) which is responsible for the swapping in and (most) of the swapping out of processes. It is a process which is never swapped and is always at the highest priority (PSWP) in the system (although it does go to sleep when it has no work to do and while the I/O for swapping processes takes place). Thus the Scheduler is a rearranger of the queues the Switcher sees even though it is in one of those queues (i.e., it may add or remove a process from consideration by swapping it out) and it is in this manner that the two functions interact..

5.4. Domain of the process Switcher and Scheduler

The process Switching function is easily isolatable to a section of the system where the choice of a new process is made. In the model of Figure 19 the boxed off area indicates the area of the model concerned by the process Switcher. Not all swapping out is performed by the Scheduler so that the Scheduling function is more distributed and is not as easily isolatable. For example, if a process requests more memory (for dynamic storage allocation or as the result of a stack overflow) the process may be automatically swapped out without any interaction with the Scheduler. The Scheduler is of course the only function capable of bringing the process back into memory but no notification of the Scheduling process is made in this case. The process is merely marked as being nonresident and placed in the CORE queue. When the Scheduler does select processes to be removed from memory, it examines the WAKEUP queue first (the processes in the WAKEUP queue are roadblocked) for a candidate. If none are found, the Switcher's queues are examined. The Scheduler is thus concerned primarily with the CORE and WAKEUP queues, although it must at times select a process from the Switcher's (runnable) queues as a candidate to boot out of memory.

6. The Scheduling Process

6.1. Operation of the Scheduler

As mentioned previously, the Scheduler is a process under UNIX. It maintains the CORE queue. When the process Switcher selects the Scheduler to run, the Scheduler will swap processes to and from the priority queues *(one at a time)*. The Scheduler will thus run in a sort of piecemeal fashion as shown in Figure 20. A typical cycle of execution is illustrated in Figure 20 where the process Switcher has selected the Scheduler to run. The Scheduler will, if need be, select a process to be swapped out to make room in memory for another process. While the I/O for the swap out is occurring, the Scheduler will go to sleep thus placing itself in the WAKEUP queue. When the I/O is complete, the Scheduler will be awakened as a result of the interrupt signalling the completion of the I/O and since it is such a high priority process (priority PSWP), it will be selected by the process Switcher almost immediately. The Scheduler will then select a process to be brought in and will again sleep until the I/O completes.

There are restrictions on the candidates to be swapped out (to prevent thrashing) so that the Scheduler rearranges memory approximately once every second. During the time that the swapping I/O occurs, the Scheduling process will pass to the WAKEUP queue until the I/O is completed. The Scheduling process will never itself enter the CORE queue as it is locked in memory.

6.2. Process Age

The process Switcher's only criteria for choosing a process to use the CPU is the priority assigned to that process. The Scheduler primarily uses a different quantity in choosing a process to swap in or out of memory. This is the *age* of a process. The age is defined as the amount of time (in seconds) that a process has spent in memory or on the swap area. The age is reset to zero whenever a process makes a transition to or from memory.

6.3. Scheduling Criteria

In selecting processes to be removed from memory, the Scheduler gives some consideration to the structure of the priority queues. The Scheduler views processes as broken into roughly two groups (see Figure 21). These are processes that will enter the priority queues at a queue whose priority is above zero, (SLEEP priorities - see Figure 21) and those that will enter the queues at some value below zero, (WAITING or USER priority queues). The queues corresponding to the SLEEP priority (this is a different concept from the sleep system call available to user processes or relinquishing the processor by the system function sleep(event, priority)) are for processes which should be given the CPU as soon as possible when awakened (to complete a system call, run the Scheduler, etc.). The WAITING priority queues are for processes that slept on an event that is not quite as urgent (blocked waiting to read a teletype, waiting for another process to terminate, etc.). The USER priority queues are those queues that a user process is in normally or is in because of a penalty and usually represent the lower priority queues of the system. When the Scheduler is looking for a process to be removed from memory it examines the WAKEUP queue and chooses the first process that is blocked at a priority that will place it in a WAITING queue when it is unblocked. If none of these processes are in the system more drastic means must be used to find memory for an incoming process. The choice of a process to bring into memory is on a *First Out First In* basis so that the oldest process on the swap area is the first to be brought in. If the oldest process on the swap area has been there for less than three seconds no further attempts to free some memory to bring it in are made. If, however, the process has been on the swap area for more than three seconds, the Scheduler searches both the Switcher's and WAKEUP queues for the process that has been in memory longest and if that process has been in memory for longer than *two seconds* it is removed regardless of its size. Note that while the Scheduler uses the priority scheme as a figure of merit for determining which process to remove from memory, the choice is primarily based on the age of the process (in memory or on the swap area) and not its priority. No attempt at compacting of memory is made by the Scheduler. (For an interesting discussion of the merits of compacting memory see [11].) A *First Available Fit* algorithm is used for placement of a swapped in processes and no consideration of memory size is given when swapping a process out. This is done to eliminate any bias toward small size processes, improving their response at the expense of larger processes. Since the granularity of age is one second, the Scheduler becomes synchronized so memory is rearranged approximately once a second and does as little work as possible in satisfying processes needs so that if there is enough memory, no swapping will occur at all.

7. Conclusion

This memorandum has served as an overview of the operation of the Scheduling and Switching functions under UNIX. A basic framework has been developed from which a more detailed discussion of the Scheduling and Switching algorithms and implementation may be given in a succeeding memorandum[17].

I would like to thank D. M. Ritchie, B.A.Tague, J.F.Maranzano and R.Brandt for reading the draft and both D. M. Ritchie and K. Thompson for their explanations of some points about the system and for their patience with my questions. Any errors in the text are the responsibility of the author.

T.M.Raleigh

MH-8234-TMR-nroff
Attachments
References(23)
Figures(21)

References

[1] Chang, W., Single-server Queuing Processes in Computing Systems, IBM Syst. J., No. 1, 1970, pp. 36-71.

[2] Coffman, E.G., and Kleinrock, L., Computer Scheduling Methods and Their Countermeasures. AFIPS SJCC 1968, Vol. 32, pp. 11-21.

[3] Coffman, E.G. and Denning, P.J., Operating Systems Theory. Prentice-Hall Inc., 1973.

[4] Digital Equipment Corporation. PDP-11/40 Processor Handbook (1972). PDP-11/45 Processor Handbook (1971). PDP-11/70 Processor Handbook (1975). PDP-11 Peripherals Handbook (1975).

[5] Hansen, P.B., Operating System Principles. Prentice Hall, Englewood N.J. 1973.

[6] Hellerman, H., Some principles of time-sharing scheduler strategies. IBM Syst. J., No. 2, 1969 pp. 94-107.

[7] Kleinrock, L., Queuing Systems. Vol. 1, John Wiley & Sons, 1975.

[8] Kleinrock, L. and Muntz, R.R., Processor Sharing Queueing Models of Mixed Scheduling Disciplines for Time Shared Systems. JACM, Vol. 19, No. 3, (July) 1972, pp. 464-482.

[9] Kleinrock, L., Time-shared Systems: A Theoretical Treatment. JACM 14, No. 2 (April) 1967, pp. 242-261.

[10] Kleinrock, L., Swap-Time Considerations in Time-Shared Systems, IEEE Transactions on Computers, June 1970, pp. 534-540.

[11] Knuth, D.E., The Art of Computer Programming: Vol. 1 Fundamental Algorithms, Addison-Wesley, pp. 435-451,1968.

[12] Lampson, B.W., A Scheduling Philosophy for Multiprocessing Systems. CACM, Vol. 11, No. 5, (May) 1968.

[13] Loren, H., Parallelism in Hardware and Software: Real and Apparent Concurrency. Prentice-Hall, Englewood Cliffs, N.J. ,1972.

[14] Madnick, S.E., and Donovan, J.J., Operating Systems. McGraw-Hill Book Company, New York 1974.

[15] McKinney, J.M., A Survey of Analytical Time-Sharing Models. Computing Surveys, Vol. 1 No. 2, (June) 1969 pp. 105-116.

[16] Organick, E. I., The Mutlics system: An examination of its structure. MIT Press Cambridge Massachusetts and London England, 1972.

[17] Raleigh, T.M., Scheduling and Switching under UNIX: Algorithms and Implementation. In preparation.

[18] Raleigh, T.M., Processes under UNIX. In preparation.

[19] Rasch, P.J., A Queueing Theory Study of Round-Robin Scheduling of Time-Shared Computer Systems. JACM, Vol. 17, No. 1, (January) 1970, pp. 131-145.

[20] Saltzer, J.E., Traffic Control in a Multiplexed Computer System. Project MAC, Thesis MAC-TR-30, June 1966.

[21] Shemer, J.E., Some Mathematical Considerations of Time-Sharing Scheduling Algorithms, JACM, Vol. 14, No. 2, (April) 1967, pp. 262-272.

[22] Thompson, K.T., and Ritchie, D.M., The UNIX time-sharing system. Comm. ACM Vol. 17 No. 7 (July) 1974 365-375.

[23] Watson, R.W., Timesharing System Design Concepts. McGraw-Hill Book Company, New York 1970.

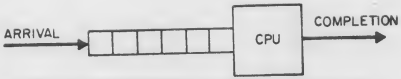

FIG. 1 PROCESSES QUEUED FOR PROCESSOR USE

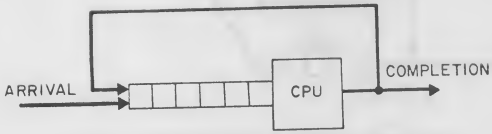

FIG. 2 FEEDBACK LOOP FOR ROUND ROBIN SERVICE

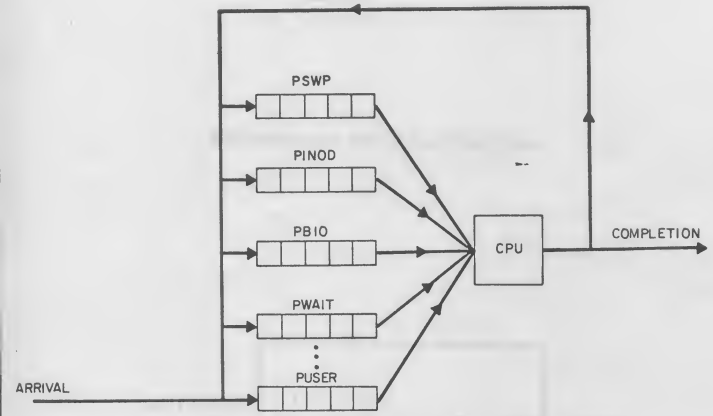

FIG. 3 ESTABLISHED QUEUES AT FIXED PRIORITY LEVELS

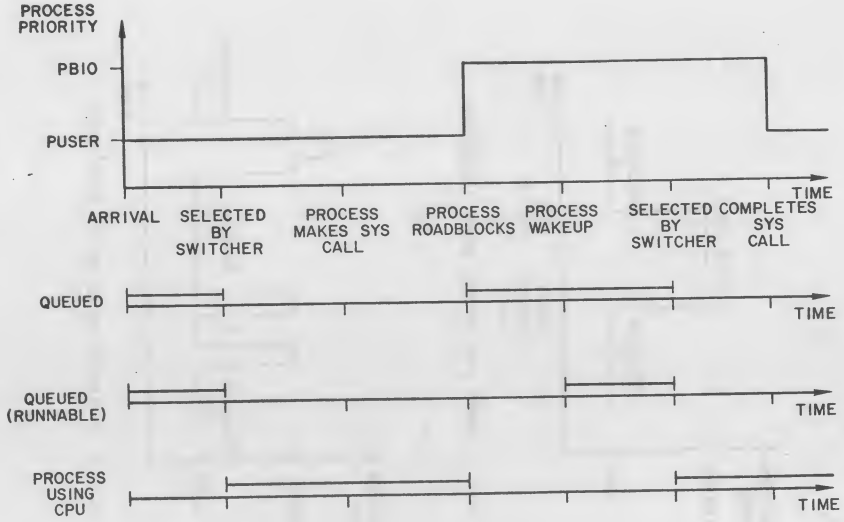

FIG. 4 EXAMPLE OF PRIORITY TRANSITIONS

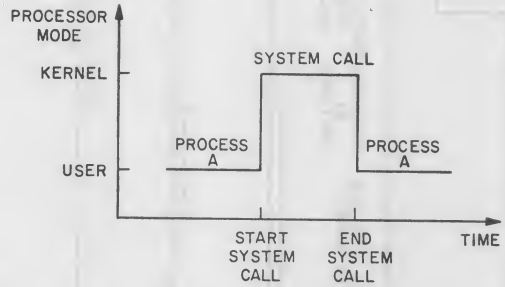

FIG. 5 USER AND KERNEL MODE EXECUTION

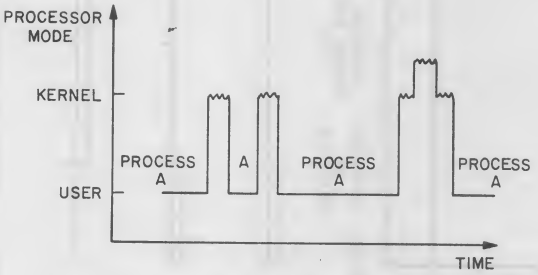

FIG. 6 INTERRUPT HANDLING IN USER MODE

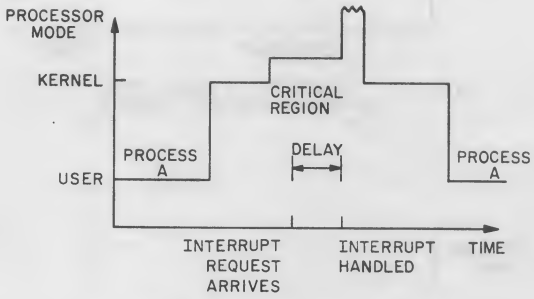

FIG. 7 CRITICAL REGIONS AND INTERRUPTS

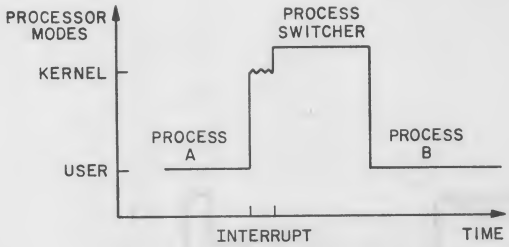

FIG. 8 INTERRUPT IN USER MODE AWAKENING ANOTHER PROCESS

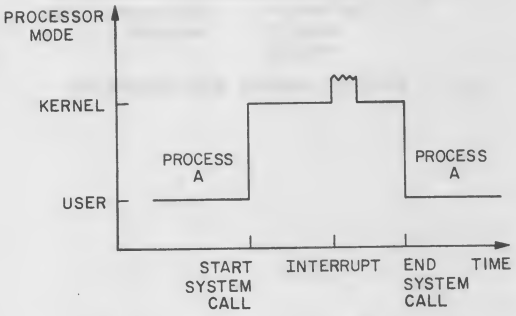

FIG. 9 INTERRUPT IN KERNEL MODE

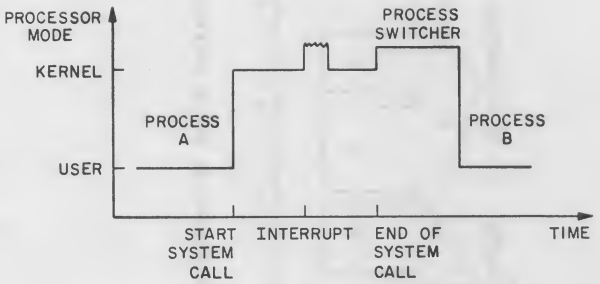

FIG. 10 INTERRUPT IN KERNEL MODE AWAKENING ANOTHER PROCESS

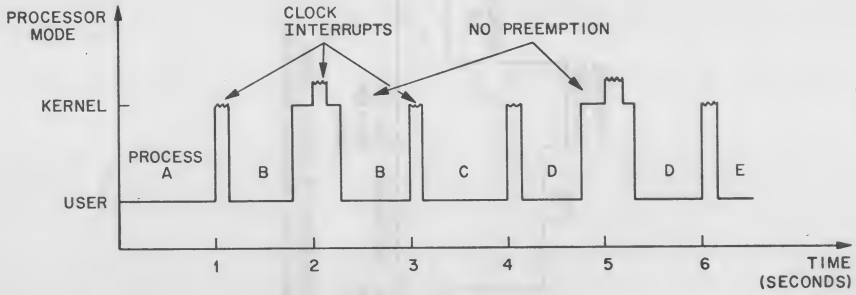

FIG. 11 TIME SLICING BY CLOCK INTERRUPT HANDLER

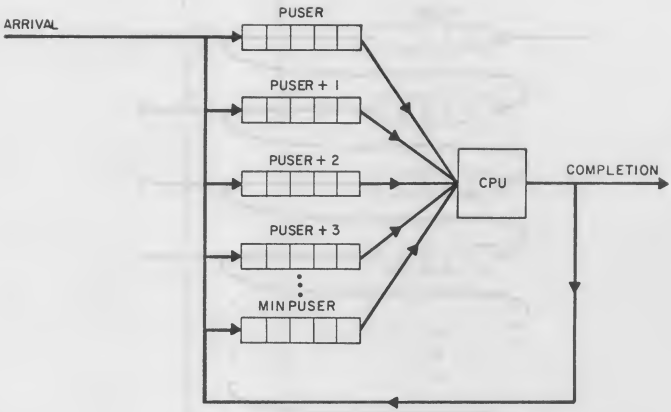

FIG. 12 QUEUES FORMED FOR USER PENALTY

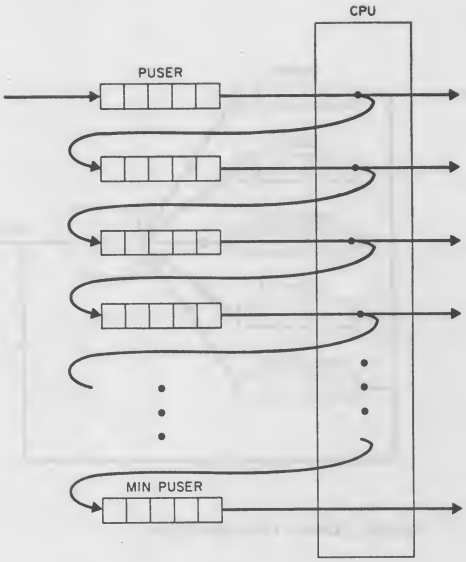

FIG. 13 SHORTEST ELAPSED TIME EFFECT

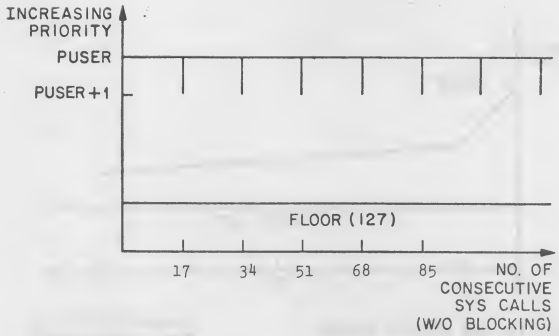

FIG. 14 PENALTY SCHEME FOR SYSTEM BOUND PROCESSES

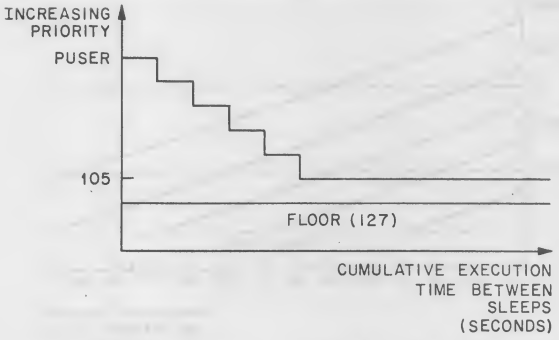

FIG. 15 PENALTY SCHEME FOR CPU BOUND PROCESSES

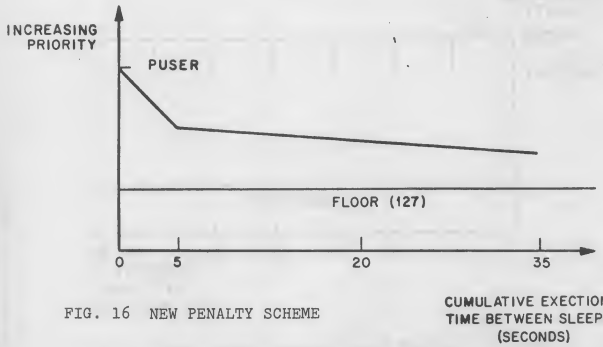

FIG. 16 NEW PENALTY SCHEME

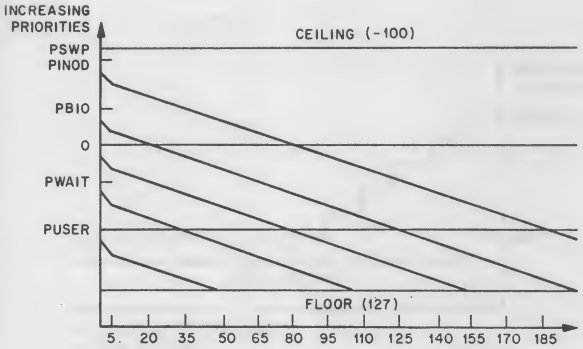

FIG. 17 EFFECT OF NICE SYSTEM CALL ON
NEW (AND OLD) PENALTY SCHEME

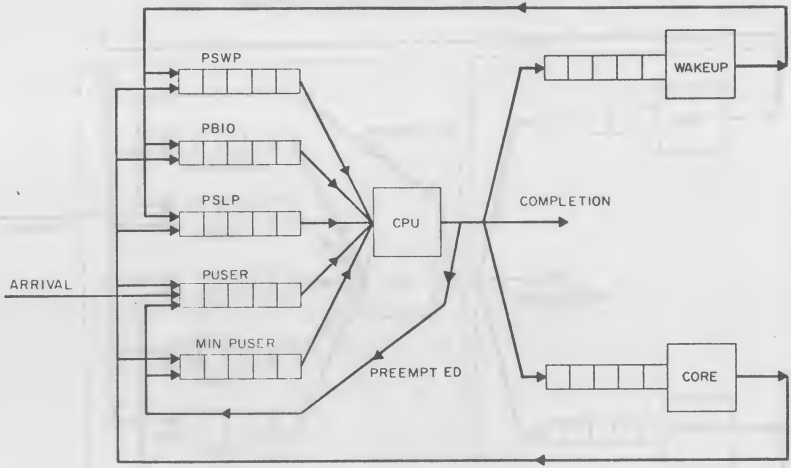

FIG. 18 SEPARATION OF SWAPPED, SLEEPING, AND RUNNABLE QUEUES

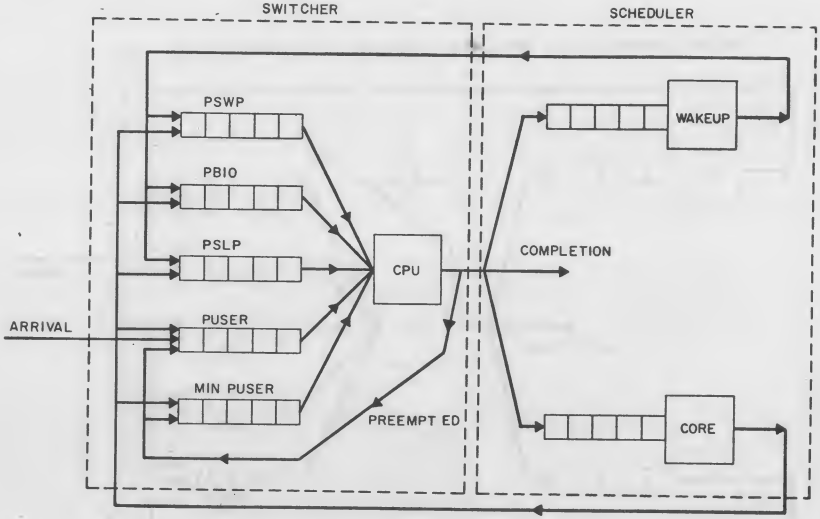

FIG. 19 DOMAIN OF THE SCHEDULER AND THE SWITCHER

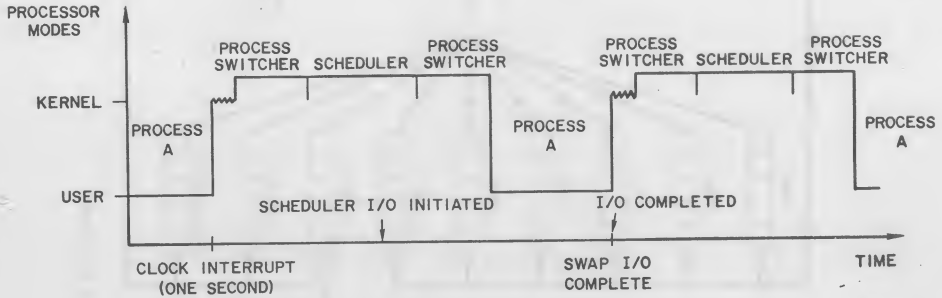

FIG. 20 INTERMITTANT OPERATION OF UNIX SCHEDULER

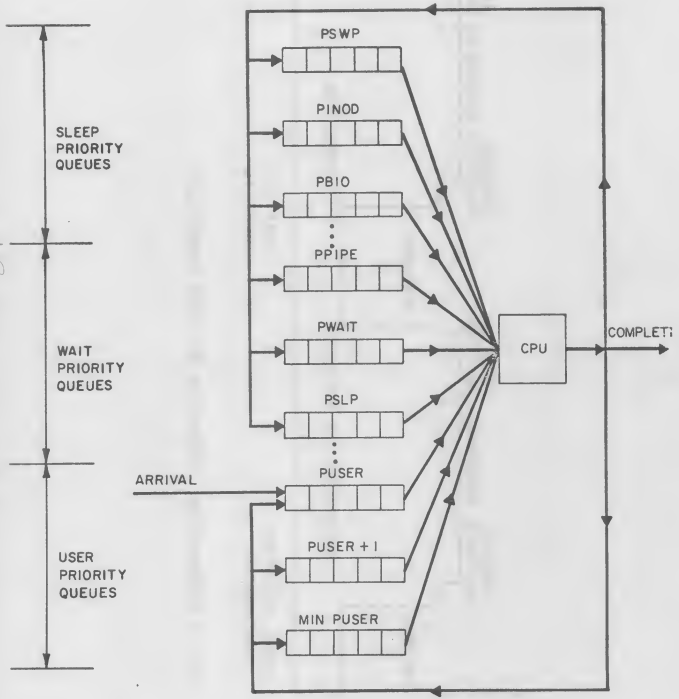

FIG. 21 GROUPING OF QUEUES